Covid-19

Požvižgale so se na prepovedi in skupaj kofetkale, telovadile ...

Takole lahko predstavimo, kdo je bil kje.

```
obiski = [("Ana", "kava"), ("Berta", "kava"), ("Cilka", "telovadba"),
          ("Dani", "zdravnik"), ("Ana", "zdravnik"), ("Cilka", "kava"),
          ("Ema", "telovadba"), ("Fanči", "telovadba"),
          ("Greta", "telovadba")]
```

Testi

Testi: testi-covid-19.py

Obvezna naloga

Napiši naslednje funkcije:

- `osebe(obiski)` prejme seznam, kot je gornji, in vrne množico imen vseh oseb, ki se pojavljajo v njih; za gornje podatke vrne `{"Ana", "Berta", "Cilka", "Dani", "Ema", "Fanči", "Greta"}`.
- `aktivnosti(obiski)` prejme seznam, kot je gornji in vrne množico vseh aktivnosti; v gornjem primeru `{"kava", "telovadba", "zdravnik"}`.
- `udelezenci(aktivnost, obiski)` prejme ime aktivnosti, na primer `"kava"` in seznam, kot je gornji, ter vrne množico imen vseh, ki so sodelovali v tej aktivnosti (torej `{"Ana", "Berta", "Cilka"}`.)
- `po_aktivnostih(obiski)` prav tako prejme seznam, vrne pa slovar, katerega ključi so aktivnosti, vrednosti pa osebe, ki so se udeleževale te aktivnosti, na primer

  ```
  {"kava": {"Ana", "Berta", "Cilka"},
   "zdravnik": {"Ana", "Dani"},
   "telovadba": {"Cilka", "Ema"}}
  ```

- `skupine(obiski)` vrne seznam množic ljudi, ki so se družile ob posameznih aktivnostih. Za gornji primer vrne `[{"Ana", "Berta", "Cilka"}, {"Ana", "Dani"}, {"Cilka", "Ema"}]`. Vrstni red množic v seznamu je lahko poljuben.
- `okuzeni(skupine, nosilci)` prejme seznam skupin, kot ga vrača prejšnja funkcija, in množico ljudi, za katere vemo, da so okuženi (`nosilci`). Funkcija vrne množico imen vseh ljudi, ki so jih `nosilci` potencialno okužili. Za skupine iz prejšnjega primera mora klic `okuzeni(skupine, {"Cilka", "Berta"})` vrniti `{"Ana", "Ema"}`.
- `zlati_prinasalec(skupine)` vrne ime osebe, ki lahko okuži največ ljudi, torej človeka, ki se je družil z največ drugimi ljudmi. Pazi, to ni nujno

tisti, ki je v največ skupinah! Kadar je enako "zlatih" oseb več, je Zlatko tisti, ki je prej po abecedi. Za gornje skupine lahko Ana in Cilka okužita po tri ljudi, zato funkcija vrne `"Ana"`.

- `korakov_do_vseh(skupine, prvi)`, ki prejme skupine in prvega okuženega. Vrne število korakov, po katerih bodo okuženi vsi. Če pa slučajno obstaja kdo (ali skupina ljudi), ki ga (jih) okužba nikoli ne doseže, ker se ni(so) družil(i) z drugimi, naj funkcija vrne `None`.

  Recimo, da imamo skupine

  ```
  [{"Cilka", "Ema", "Jana", "Saša"},
  {"Ema"},
  {"Fanči", "Greta", "Saša"},
  {"Greta", "Nina"},
  {"Greta", "Olga", "Rebeka"},
  {"Micka", "Ana", "Klara"},
  {"Fanči", "Iva", "Berta", "Špela"},
  {"Klara", "Cilka", "Dani"},
  {"Petra", "Dani", "Lara", "Špela"}]
  ```

  in da je prvookužena Ana. Ana bo okužila Micko in Klaro. V drugem koraku bosta tidve okužili Dani in Cilko. V tretjem koraku bosta tidve okužili Laro, Jano, Petro, Sašo, Emo in Špelo. V četrtem koraku bodo te okužile Berto, Fanči, Greto in Ivo. V petek bodo le-te okužile Olgo, Nino in Rebeko. Da bodo bolni vsi, bo torej potrebnih pet korakov - če začnemo z Ano. Funkcija zato v tem primeru vrne 5.

korak	okuženi
0	Ana
1	Micka, Klara
2	Dani, Cilka
3	Lara, Jana, Petra, Saša, Ema, Špela
4	Berta, Fanči, Greta, Iva
5	Olga, Nina, Rebeka

Dodatna naloga

Vse funkcije razen zadnjih treh (`okuženi`, `zlati_prinasalec`, `korakov_do_vseh`) napiši s pomočjo izpeljanih seznamov, množic, slovarjev, tako da bodo dolge le eno vrstico - vsebovale bodo samo `return ...`. Pri tem seveda ne smeš definirati dodatnih funkcij, saj bi lahko potem goljufal tako, da bi vse delo opravil v njih.

Izziv

Naštudiraj funkcijo `functools.reduce` dodatne argumente funkcij, kot so `min`, `max` ali `sort` ... in v eni vrstici napiši vse funkcije razen `korakov_do_vseh`. (`reduce` seveda uporabi, kjer ga potrebuješ. Kjer ne, ne.)

Rešitev

Prve tri so res preproste. Zahtevajo samo, da znamo sestaviti slovar.

`osebe`, `aktivnosti` in `udelezenci`

```
def osebe(obiski):
    vse_osebe = set()
    for oseba, aktivnost in obiski:
        vse_osebe.add(oseba)
    return vse_osebe

def aktivnosti(obiski):
    vse_aktivnosti = set()
    for oseba, aktivnost in obiski:
        vse_aktivnosti.add(aktivnost)
    return vse_aktivnosti

def udelezenci(aktivnost, obiski):
    vsi_udelezenci = set()
    for oseba, aktivnost1 in obiski:
        if aktivnost1 == aktivnost:
            vsi_udelezenci.add(oseba)
    return vsi_udelezenci
```

Hitro jih še preskusimo.

```
osebe(obiski)
```

```
{'Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fanči', 'Greta'}
```

```
aktivnosti(obiski)
```

```
{'kava', 'telovadba', 'zdravnik'}
```

```
udelezenci("kava", obiski)
```

```
{'Ana', 'Berta', 'Cilka'}
```

Tudi v izpeljane sezname jih pretvorimo, ne da bi zašvicali.

```
def osebe(obiski):
    return {oseba for oseba, aktivnost in obiski}

def aktivnosti(obiski):
```

```
    return {aktivnost for oseba, aktivnost in obiski}

def udelezenci(aktivnost, obiski):
    return {oseba for oseba, aktivnost1 in obiski if aktivnost1 == aktivnost}
```

po_aktivnostih in skupine

Funkcija `po_aktivnostih` je preprosta, če uporabimo funkciji `aktivnosti` in `udeleženci`.

```
def po_aktivnostih(obiski):
    po_akt = {}
    for aktivnost in aktivnosti(obiski):
        po_akt[aktivnost] = udelezenci(aktivnost, obiski)
    return po_akt
```

Brez njiju nas čaka tole:

```
from collections import defaultdict

def po_aktivnostih(obiski):
    po_akt = defaultdict(set)
    for oseba, aktivnost in obiski:
        po_akt[aktivnost].add(oseba)
    return po_akt
```

Ali, če se ne domislimo `defaultdict`-a, celo tole:

```
def po_aktivnostih(obiski):
    po_akt = {}
    for oseba, aktivnost in obiski:
        if aktivnost not in po_akt:
            po_akt[aktivnost] = set()
        po_akt[aktivnost].add(oseba)
    return po_akt
```

Nobena od teh dveh tudi ne bo voljna zlesti v eno vrstico, v izpeljan slovar. Bistvena razlika med tema dvema funkcijama in prvo je, da tidve dodajata v slovar: vanj vstavita nek ključ in vrednost, kasneje pa k tej vrednosti še dodajata. Izpeljani slovarji pa so "deklarativni": povemo, kaj je ključ in kakšna je pripadajoča vrednost. Kasnejšega dodajanja ni.

Prva pa se kar sama ponuja, da bi šla v eno vrstico.

```
def po_aktivnostih(obiski):
    return {aktivnost: udelezenci(aktivnost, obiski) for aktivnost in aktivnosti(obiski)}
```

Preverimo, da res deluje.

```
po_aktivnostih(obiski)
```

```
{'zdravnik': {'Ana', 'Dani'},
 'telovadba': {'Cilka', 'Ema', 'Fanči', 'Greta'},
 'kava': {'Ana', 'Berta', 'Cilka'}}
```

Funkcija `skupine` pa je najpreprostejša od vseh: zahteva le, da vrnemo vse vrednosti v slovarju, kakršnega sestavi `po_aktivnostih`.

```
def skupine(obiski):
    return list(po_aktivnostih(obiski).values())
```

```
skupine(obiski)
```

```
[{'Ana', 'Dani'},
 {'Cilka', 'Ema', 'Fanči', 'Greta'},
 {'Ana', 'Berta', 'Cilka'}]
```

Bi se dalo drugače, brez te funkcije? Da, vendar ne brez slovarja. Vsaj ne prav lahko. Vsaka razumna rešitev, ki ne bi klicala funkcije `aktivnosti`, bi v bistvu ponavljala, kar smo napisali v eni od gornjih oblik te funkcije.

`okuzeni` in `zlati_prinasalec`

Funkcija `okuzeni` od študenta zahteva, da zna delati z množicami. Konkretno, računati preseke, unije in razlike. Iti mora čez vse skupine. Za vsako skupino preveri, ali je v njej kdo okužen (tako da izračuna *presek* članov te skupine in tistih, ki so bolni). Če je, potem mora člane te skupine dodati (*unija*) k novookuženim. Ker funkcija ne sme vrniti tudi imen tistih, ki so bili okuženi že od prej, mora na koncu od množice novookuženih odšteti množico tistih, ki so bili okuženi že prej.

```
def okuzeni(skupine, nosilci):
    vsi_okuzeni = set()
    for skupina in skupine:
        if skupina & nosilci:
            vsi_okuzeni |= skupina
    return vsi_okuzeni - nosilci
```

```
skupine = [{'Ana', 'Dani'},
           {'Ana', 'Berta', 'Cilka'},
           {'Cilka', 'Ema', 'Fanči', 'Greta'}]
```

```
okuzeni(skupine, {"Dani", "Berta"})
```

```
{'Ana', 'Cilka'}
```

Reševanje te naloge v eni vrstici je sodilo pod dodaten izziv. Mislil sem namreč, da bo potrebno zanj uporabiti funkcijo `reduce`. Poleg tega pa še uporabiti `set.union`, za katero niti ne vemo dobro, kaj je (v resnici: nevezana metoda, *unbound method*, kar je v sodobnem Pythonu praktično isto kot funkcija).

```
from functools import reduce

def okuzeni(skupine, nosilci):
    return reduce(set.union,
                  (skupina for skupina in skupine if skupina & nosilci),
                  set()
                 ) - nosilci
```

`reduce` se vam gotovo splača naštudirati. Eden od študentov pa je odkril, da `set.union` kot argument sprejme tudi poljubno število množi. To rešitev poenostavi v

```
def okuzeni(skupine, nosilci):
    return set.union(*(skupina for skupina in skupine if skupina & nosilci), set()) - nosilc
```

Za `zlatega_prinasalca` najprej potrebujemo seznam oseb. Potem pa za vsako od oseb preveri, koliko jih okuži in si zapomnimo tisto, ki jih okuži največ. To je preprosto, če se spomnimo (in znamo) uporabiti funkcijo `okuzeni`.

```
def zlati_prinasalec(skupine):
    vse_osebe = set()
    for skupina in skupine:
        vse_osebe |= skupina

    zlatko = None
    zrtev = 0
    for oseba in vse_osebe:
        n_okuzenih = len(okuzeni(skupine, {oseba}))
        if n_okuzenih > zrtev or (n_okuzenih == zrtev and oseba < zlatko):
            zlatko = oseba
            zrtev = n_okuzenih
    return zlatko
```

```
zlati_prinasalec(skupine)
```

```
'Cilka'
```

Največji kamen spotike je bil, kako klicati funkcijo `okuzeni` z eno samo osebo. Klic `okuzeni(skupine, oseba)` ne bo deloval, ker `okuzeni` kot drugi argument pričakuje množico, `oseba` pa je niz. Klic `okuzeni(skupine, set(oseba))`, ker `set("Cilka")` ni množica, ki vsebuje niz `Cilka`, temveč

```
set("Cilka")
```

```
{'C', 'a', 'i', 'k', 'l'}
```

Množico, ki vsebuje niz `"Cilka"` zapišemo kot `{"Cilka"}`, množico, ki vsebuje niz, shranjen v spremenljivki `oseba`, pa kot `{oseba}`. Pravilni klic je torej takšen, kot ga vidimo zgoraj: `okuzeni(skupine, {oseba})`.

V eno vrstico - kar je bil spet le dodani izziv - ga najlažje spravimo tako, da

sestavimo pare (-število_okuženih, ime) in poiščemo minimum. Na ta način bomo dobili tistega, ki jih je okužil največ (ker iščemo minimum negativne vrednosti), med tistimi, ki so jih okužili enako, pa tistega, ki je prej po abecedi. Če bi iskali maksimum parov (število okuženih, ime), bi med enako pridnimi prenašalci dobili tistega, ki je po abecedi kasneje.

Tudi s tem trikom rešitev v eni vrstici zahteva lambdo.

```
def zlati_prinasalec(skupine):
    return min(set.union(*skupine), key=lambda oseba: (-len(okuzeni(skupine, {oseba})), oseb
```

Gre tudi brez, a ni nič lepše.

```
def zlati_prinasalec(skupine):
    return min((-len(okuzeni(skupine, {oseba})), oseba) for oseba in set.union(*skupine))[1]
```

korakov_do_vseh

Tale pa je študentom nepričakovano dala vetra.

Bistvo funkcije je, da širimo množico okuženih. V bistvu delamo `vsi_okuzeni |= okuzeni(skupine, vsi_okuzeni)` - k vsem okuženim v vsakem koraku dodamo vse, ki jih okužijo vsi, ki so okuženi.

V začetku potrebujemo množico vseh oseb. To lahko dobimo tako, kot v `zlati_prinasalec`, ali po bližnjici, ki smo jo odkrili zgoraj, `vsi = set.union(*skupine)`. Množica prvotno okuženih je `{prinasalec}`.

Nato potrebujemo zanko, eno in edino. V njej bomo šteli korake. Zanka je lahko oblike

```
korakov = 0
while True:
    ...
    korakov += 1
```

po prejšnjih predavanjih pa vemo za `itertools.count()`, ki je v bistvu neskončen `range`, torej funkcija, ki šteje od 0 do neskončno. Gornjo zanko lahko torej zamenjamo z

```
for korakov in count():
```

Znotraj znake pa nam je ponavljati tole. Če so `doslej_okuzeni` že `vsi`, vrnemo število korakov. (To je lahko tudi `0`, če je `prinasalec` edina oseba na svetu. Sicer izvemo, kdo so `novookuzeni`. Če jih ni, vrnemo `None`, ker še niso okuženi vsi, in tudi nikoli ne bodo. Če imamo kakega novookuženega, pa ga dodamo k doslej okuženim.

```
from itertools import count

def korakov_do_vseh(skupine, prinasalec):
    vsi = set.union(*skupine)
```

```
    doslej_okuzeni = {prinasalec}
    for i in count():
        if doslej_okuzeni == vsi:
            return i
        novookuzeni = okuzeni(skupine, doslej_okuzeni)
        print(novookuzeni)
        if not novookuzeni:
            return None

        doslej_okuzeni |= novookuzeni
```

Za primerjavo s tabelico iz opisa naloge, smo dodali še en `print`

```
korakov_do_vseh([{"Cilka", "Ema", "Jana", "Saša"},
    {"Ema"},
    {"Fanči", "Greta", "Saša"},
    {"Greta", "Nina"},
    {"Greta", "Olga", "Rebeka"},
    {"Micka", "Ana", "Klara"},
    {"Fanči", "Iva", "Berta", "Špela"},
    {"Klara", "Cilka", "Dani"},
    {"Petra", "Dani", "Lara", "Špela"}], "Ana")
```

```
{'Klara', 'Micka'}
{'Cilka', 'Dani'}
{'Ema', 'Lara', 'Saša', 'Špela', 'Petra', 'Jana'}
{'Iva', 'Berta', 'Greta', 'Fanči'}
{'Nina', 'Rebeka', 'Olga'}
```

```
5
```

Točno, kot mora biti:

korak	okuženi
0	Ana
1	Micka, Klara
2	Dani, Cilka
3	Lara, Jana, Petra, Saša, Ema, Špela
4	Berta, Fanči, Greta, Iva
5	Olga, Nina, Rebeka

Nekatere je zanimala še rešitev te naloge v eni vrstici - tega nisem namreč pisal niti v izziv.

Osnovni problem je, kako shraniti nekaj, kar si naračunal, v spremenljivko, da ne bo potrebno računati ponovno. Python 3.8 ima "walrus operator". Meni se zdi ogaben in ga nisem še nikoli uporabil. Spodnji trik ima vsaj neko lispovsko

eleganco (spominja na `let`, ki ga imamo tudi v kotlinu): če hočeš neko naračunano vrednost uporabiti večkrat, napišeš lambdo in jo pokličeš s to vrednostjo. Se pravi, namesto `f(x) * (1 + f(x))` napišeš `(lambda y: y * (1 + y))(f(x))`.

To naredi spodnjo funkcijo skoraj prebavljivo.

```
def korakov_do_vseh(skupine, doslej):
    return korakov_do_vseh(skupine, {doslej}) if isinstance(doslej, str) else (
        0 if set.union(*skupine) == doslej else
        (lambda novi:
             (lambda korakov: None if korakov is None else korakov + 1)(korakov_do_vseh(sku
        (okuzeni(skupine, doslej))
    )
```